JOYCE—A PROGRAMMING LANGUAGE FOR DISTRIBUTED SYSTEMS

PER BRINCH HANSEN

(1987)

This paper describes a secure programming language called Joyce based on CSP and Pascal. Joyce permits unbounded (recursive) activation of communicating agents. The agents exchange messages through synchronous channels. A channel can transfer messages of different types between two or more agents. A compiler can check message types and ensure that agents use disjoint sets of variables only. The use of Joyce is illustrated by a variety of examples.

1 INTRODUCTION

Two years after the invention of the monitor concept (Brinch Hansen 1973; Hoare 1974), Concurrent Pascal had been developed (Brinch Hansen 1975) and used for operating system design (Brinch Hansen 1976). Within ten years, half a dozen production-quality languages were monitor-based, among them Modula (Wirth 1977), Pascal-Plus (Welsh 1979), Mesa (Lampson 1980) and Concurrent Euclid (Holt 1982).

Eight years after the CSP proposal (Hoare 1978), several CSP-based languages have been developed: these include CSP80 (Jazayeri 1980), RBCSP (Roper 1981), ECSP (Baiardi 1984), Planet (Crookes 1984) and the low-level language occam (Inmos 1984). But no experience has been reported on the use of these languages for non-trivial system implementation. Although CSP has been highly successful as a notation for theoretical work

P. Brinch Hansen, Joyce—A programming language for distributed systems. *Software—Practice and Experience 17* , 1 (January 1987), 29–50.

(Hoare 1985), it has probably been too far removed from the requirements of a secure programming language.

This paper describes a secure programming language called Joyce for the design and implementation of distributed systems. Joyce is based on CSP and Pascal (Wirth 1971).

A Joyce program consists of nested procedures which define communicating agents. Joyce permits unbounded (recursive) activation of agents. The execution of a program activates an initial agent. Agents may dynamically activate subagents which run concurrently with their creators. The variables of an agent are inaccessible to other agents.

Agents communicate by means of symbols transmitted through channels. Every channel has an alphabet—a fixed set of symbols that can be transmitted through the channel. A symbol has a name and may carry a message of a fixed type.

Two agents match when one of them is ready to output a symbol to a channel and the other is ready to input the same symbol from the same channel. When this happens, a communication takes place in which a message from the sending agent is assigned to a variable of the receiving agent.

The communications on a channel take place one at a time. A channel can transfer symbols in both directions between two agents.

A channel may be used by two or more agents. If more than two agents are ready to communicate on the same channel, it may be possible to match them in several different ways. The channel arbitrarily selects two matching agents at a time and lets them communicate.

A polling statement enables an agent to examine one or more channels until it finds a matching agent. Both sending and receiving agents may be polled.

Agents create channels dynamically and access them through local port variables. When an agent creates a channel, a channel pointer is assigned to a port variable. The agent may pass the pointer as a parameter to subagents.

When an agent reaches the end of its defining procedure, it waits until all its subagents have terminated before terminating itself. At this point, the local variables and any channels created by the agent cease to exist.

This paper defines the concepts of Joyce and illustrates the use of the language to implement a variety of well-known programming concepts and algorithms.

2 LANGUAGE CONCEPTS

Joyce is based on a minimal Pascal subset: type integer, boolean, char and real; enumerated, array and record types; constants, variables and expressions; assignment, if, while, compound and empty statements.

This subset is extended with concurrent programming concepts called agent procedures, port types and channels, agent, port, input/output and polling statements.

The Joyce grammar is defined in extended BNF notation: $[E]$ denotes an E sentence (or none). $\{E\}$ denotes a finite (possibly empty) sequence of E sentences. Tokens are enclosed in quotation marks, e.g. "**begin**".

This paper concentrates on the concurrent aspects of Joyce.

Port types

TypeDefinition = TypeName "=" NewType ";" .
NewType = PascalType | PortType .
PortType = "[" Alphabet "]" .
Alphabet = SymbolClass { "," SymbolClass } .
SymbolClass = SymbolName ["(" MessageType ")"] .
MessageType = TypeName .

A Joyce program defines abstract concurrent machines called agents. The agents communicate by means of values called symbols transmitted through entities called channels. The set of possible symbols that can be transmitted through a channel is called its alphabet.

Agents create channels dynamically and access them through variables known as port variables. The types of these variables are called port types.

A type definition

$$T = [s_1(T_1), s_2(T_2), \ldots, s_n(T_n)];$$

defines a port type named T. The port value *nil* T is of type T and denotes a non-existing channel. All other port values of type T denote distinct channels with the given alphabet. The port values (also known as channel pointers) are unordered.

The alphabet is the union of a fixed number of disjoint symbol classes named $s_1, s_2, \ldots, s_n$.

A symbol class $s_i(T_i)$ consists of every possible value of type T_i prefixed with the name s_i. The T_i values are called messages.

A symbol class s_j consists of a single symbol named s_j without a message. The symbol is called a signal.

The symbol names $s_1, s_2, \ldots, s_n$ must be distinct, and $T_1, T_2, \ldots, T_n$ must be names of known types. (Every type has a name and is said to be known within its scope.) The message types cannot be (or include) port types.

Examples:

1. A port type named *stream* with two symbol classes named *int* and *eos*. Every *int* symbol includes a message of type integer. The *eos* symbol is a signal:

```
stream = [int(integer), eos];
```

2. A port type named PV with two signals P and V:

```
PV = [P, V];
```

Note. Symbols of the same alphabet must have distinct names. Symbols of different alphabets may have the same names. Different symbols of the same alphabet may carry messages of the same type.

Port variables

```
PortAccess = VariableAccess .
```

A variable $v : T$ of a port type T holds a port value. If the value of v is *nil* T, a port access v denotes a non-existing channel; otherwise, it denotes a channel with the alphabet given by T. (The channel itself is not a variable, but a communication device shared by agents.)

Examples:

1. Access a port variable named *inp*:

```
inp
```

2. Access the ith element of an array of port variables named *ring*:

```
ring[i]
```

Port statements

Statement = PascalStatement | PortStatement |
InputOutputStatement | PollingStatement |
AgentStatement .
PortStatement = "+" PortAccess .

The creation of a new channel is called the activation of the channel. A port statement $+c$ denotes activation of a new channel. The variable access c must be of a known port type T.

When an agent executes the port statement, a new channel with the alphabet given by T is created and a pointer to the channel is assigned to the port variable c. The agent is called the creator of the channel. The channel itself is known as an internal channel of the agent. The channel ceases to exist when its creator terminates.

Examples:

1. Create a new channel and assign the pointer to the port variable *inp*:

```
+inp
```

2. Create a new channel and assign the pointer to the port variable *ring[i]*:

```
+ring[i]
```

Input/output statements

InputOutputCommand = OutputCommand | InputCommand .
OutputCommand = PortAccess "!" OutputSymbol .
OutputSymbol = SymbolName ["(" OutputExpression ")"] .
OutputExpression = Expression .
InputCommand = PortAccess "?" InputSymbol .
InputSymbol = SymbolName ["(" InputVariable ")"] .
InputVariable = VariableAccess .
InputOutputStatement = InputOutputCommand .

A communication is the transfer of a symbol from one agent to another through a channel. The sending agent is said to output the symbol, and the

receiving agent is said to input the symbol. The agents access the channel through local port variables.

Consider an agent p which accesses a channel through a port variable b, and another agent q which accesses the same channel through a different port variable c. The port variables must be of the same type:

$$T = [s_1(T_1), s_2(T_2), \ldots, s_n(T_n)];$$

An output command $b!s_i(e_i)$ denotes output of a symbol $s_i(e_i)$ through the channel denoted by the port variable b. s_i must be the name of one of the symbol classes of T, and the expression e_i must be of the corresponding message type T_i.

An input command $c?s_i(v_i)$ denotes input of a symbol $s_i(v_i)$ through the channel denoted by the port variable c. s_i must be the name of one of the symbol classes of T, and the variable access v_i must be of the corresponding message type T_i.

When an agent p is ready to output the symbol s_i on a channel, and another agent q is ready to input the same symbol from the same channel, the two agents are said to match and a communication between them is said to be feasible. If and when this happens, the two agents execute the output and input commands simultaneously. The combined effect is defined by the following sequence of actions:

1. p obtains a value by evaluating the output expression e_i.
2. q assigns the value to its input variable v_i.

(If the symbol s_i is a signal, steps 1 and 2 denote empty actions.)

After a communication, the agents proceed concurrently.

When an agent reaches an input/output command which denotes a communication that is not feasible, the behavior of the agent depends on whether the command is used as an input/output statement or as a polling command (defined in the next section).

The effect of an input/output statement is to delay an agent until the communication denoted by the statement has taken place.

The communications on a channel take place one at a time. A channel can transfer symbols in both directions between two agents.

A channel may be used by two or more agents. If more than two agents are ready to communicate on the same channel, it may be possible to match them in several different ways. The channel arbitrarily selects two matching agents at a time and lets them communicate.

Examples:

1. Use the port variable *out* to output an *int* symbol with the message $x + 1$:

   out!int(x + 1)

2. Use the port variable *inp* to input an *int* symbol and assign the message to y:

   inp?int(y)

3. Use the port variable *out* to output an *eos* signal:

   out!eos

4. Use the port variable *inp* to input an *eos* signal:

   inp?eos

5. Use the port variable *ring[i]* to output a *token* signal:

   ring[i]!token

Polling statements

PollingStatement =
  "**poll**" GuardedStatementList "**end**" .
GuardedStatementList =
  GuardedStatement { "|" GuardedStatement } .
GuardedStatement = Guard "−>" StatementList .
Guard = PollingCommand ["&" PollingExpression] .
PollingCommand = InputOutputCommand .
PollingExpression = BooleanExpression .

A polling statement

poll
  C_1 & B_1 -> SL_1 |
  C_2 & B_2 -> SL_2 |
    ...
  C_n & B_n -> SL_n
end

denotes execution of exactly one of the guarded statements

$$C_i \ \& \ B_i \ -> \ SL_i$$

An agent executes a polling statement in two phases, known as the polling and completion phases:

1. Polling: the agent examines the guards $C_1\&B_1$, $C_2\&B_2$, ..., $C_n\&B_n$ cyclically until finds one with a polling command C_i that denotes a feasible communication and a polling expression B_i that denotes true (or is omitted).

2. Completion: the agent executes the selected polling command C_i followed by the corresponding statement list SL_i.

While an agent is polling, it can be matched only by another agent that is ready to execute an input/output statement. Two agents polling at the same time do not match.

Example:

Use a port variable named *user* to either (1) input a P signal (provided an integer $x > 0$) and decrement x, or (2) input a V signal and increment x:

```
poll
  user?P & x > 0 -> x := x - 1 |
  user?V -> x := x + 1
end
```

Note. Polling has no side-effects, but may cause program failure if the expression evaluation causes a range error (or overflow).

Agent statements

AgentStatement =
  AgentName ["(" ActualParameterList ")"] .
ActualParameterList =
  ActualParameter { "," ActualParameter } .
ActualParameter = Expression .

An agent procedure P defines a class of agents. The creation and start of an agent is called its activation. The activation of a P agent creates a new instance of every variable defined in procedure P. These variable instances are called the own variables of the new agent. When the agent refers to a variable x in P, it refers to its own instance of x. The own variables of an agent are inaccessible to other agents.

An agent is always activated by another agent (called its creator). The new agent is called a subagent of its creator. After the creation, the subagent and its creator run concurrently.

An agent statement

$$P(e_1, e_2, \ldots, e_m)$$

denotes activation of a new agent. P must be the name of a known agent procedure (defined in the next section). The actual parameter list must contain an actual parameter e_i for every formal parameter a_i defined by P. e_i must be an expression of the same type as a_i.

When an agent executes an agent statement, a subagent is created in two steps:

1. The own variables of the subagent are created as follows:
   (a) The formal parameters of P are created one at a time in the order listed. Every formal parameter a_i is assigned the value denoted by the corresponding actual parameter e_i.
   (b) The variables defined in the procedure body of P are created with unpredictable initial values.
2. The subagent is started.

A port operand used as an actual parameter denotes a channel which is accessible to both the subagent and its creator. It is known as an external channel of the subagent.

An agent defined by a procedure P may activate P recursively. Every activation creates a new P agent with its own variables.

Example:

Activate a semaphore agent with two actual parameters: the integer 1 and a port value named *user*:

```
semaphore(1, user)
```

Agent procedures

```
AgentProcedure = "agent" AgentName ProcedureBlock ";" .
ProcedureBlock =
  [ "(" FormalParameterList ")" ] ";" ProcedureBody .
FormalParameterList =
  ParameterDefinition { ";" ParameterDefinition } .
ParameterDefinition =
  VariableName { "," VariableName } ":" TypeName .
ProcedureBody =
  [ ConstantDefinitionPart ] [ TypeDefinitionPart ]
  { AgentProcedure } [ VariableDefinitionPart ]
  CompoundStatement .
```

An agent procedure P defines a class of agents. Every formal parameter is a local variable that is assigned the value of an expression when a P agent is activated.

After its activation, a P agent executes the corresponding procedure body in two steps:

1. The agent executes the compound statement of P.
2. The agent waits until all its subagents (if any) have terminated. At this point, the own variables and internal channels of the agent cease to exist, and the agent terminates.

Example: semaphore

An agent procedure that defines a semaphore which accepts P and V signals:

```
agent semaphore(x: integer; user: PV);
begin
  while true do
    poll
      user?P & x > 0 -> x := x − 1|
      user?V -> x := x + 1
    end
end;
```

Programs

```
Program =
  [ ConstantDefinitionPart ][ TypeDefinitionPart ]
  AgentProcedure .
```

A program defines an agent procedure P. The program is executed by activating and executing a single P agent (the initial agent). The activation of the initial agent is the result of executing an agent statement in another program (an operating system). A program communicates with its operating system through the external channels of the initial agent (the system channels).

3 PROGRAM EXAMPLES

The following examples illustrate the use of Joyce to implement stream processing, functions, data representations, monitors and ring nets. The examples have been compiled and run on an IBM PC using a Joyce compiler and interpreter written in Pascal.

Stream processing

First, we look at agents that input and output bounded data streams. Every stream is a (possibly empty) sequence of integers ending with an *eos* signal:

```
type stream = [int(integer), eos];
```

Example: generate

An agent that generates an arithmetic progression $a_0, a_1, \ldots, a_{n-1}$, where $a_i = a + i \times b$:

```
agent generate(out: stream;
  a, b, n: integer);
```

```
var i: integer;
begin
  i := 0;
  while i < n do
    begin
      out!int(a + i*b); i := i + 1
    end;
  out!eos
end;
```

Example: copy

An agent that copies a stream:

```
agent copy(inp, out: stream);
var more: boolean; x: integer;
begin
  more := true;
  while more do
    poll
      inp?int(x) -> out!int(x)|
      inp?eos -> more := false
    end;
  out!eos
end;
```

Example: merge

An agent that outputs an arbitrary interleaving of two input streams:

```
agent merge(inp1, inp2, out: stream);
var n, x: integer;
begin
  n := 0;
  while n < 2 do
    poll
      inp1?int(x) -> out!int(x)|
      inp1?eos -> n := n + 1|
      inp2?int(x) -> out!int(x)|
      inp2?eos -> n := n + 1
```

```
    end;
  out!eos
end;
```

A value input from one of the streams inp_1 and inp_2 is immediately output. The agent terminates when both input streams have been exhausted ($n = 2$).

Example: suppress duplicates

An agent that outputs a stream derived from an ordered input stream by suppressing duplicates:

```
agent suppress(inp, out: stream);
var more: boolean; x, y: integer;
begin
  poll
    inp?int(x) -> more := true|
    inp?eos -> more := false
  end;
  while more do
    poll
      inp?int(y) ->
        if x <> y then
          begin out!int(x); x := y end|
      inp?eos -> out!int(x); more := false
    end;
  out!eos
end;
```

Example: iterative buffer

A buffer implemented as a pipeline of 10 copy agents:

```
agent buffer(inp, out: stream);
const n = 9;
type net = array [1..n] of stream;
use copy;
var a: net; i: integer;
begin
  +a[1]; copy(inp, a[1]); i := 2;
```

```
    while i <= n do
      begin
        +a[i]; copy(a[i−1], a[i]); i := i + 1
      end;
    copy(a[n], out)
  end;
```

The buffer agent is a composite agent which activates an array of copy agents and channels by iteration. The length $n+1$ of the iterative array is specified by a constant n. During compilation, the *use* sentence is replaced by the text of the copy agent.

This algorithm is an example of "information hiding". A user agent may regard the copy and buffer agents as different implementations of the same mechanism: a copying agent with an input and an output channel. The subagents and internal channels of the buffer agent are therefore made invisible to its environment.

Example: recursive buffer

A recursive version of the previous buffer:

```
agent buffer(n: integer; inp, out: stream);
use copy;
var succ: stream;
begin
  if n = 1 then copy(inp, out)
  else
    begin
      +succ; copy(inp, succ);
      buffer(n − 1, succ, out)
    end
end;
```

The length n of the recursive array is specified when it is activated. If $n = 1$, the buffer consists of a single copy agent only; otherwise, it consists of a copy agent followed by a buffer of length $n - 1$.

The next two examples illustrate the use of a programming paradigm known as a dynamic accumulator. This is a pipeline which uses an input stream to compute another stream. The pipeline accumulates the new stream while it is being computed and outputs it as a whole when it is complete. Every agent (except the last one) in the pipeline holds one element of

the new stream. The last agent is empty. Each time the pipeline has computed another element, the last agent receives an element and extends the pipeline with a new empty agent. Since the length of the computed stream is not known *a priori*, the pipeline begins as a single empty agent. At the end of the input stream, the pipeline outputs the elements of the computed stream one at a time and terminates.

Example: recursive sorting

A dynamic accumulator that inputs a (possibly empty) stream and outputs the elements in non-decreasing order:

```
agent sort(inp, out: stream);
var more: boolean; x, y: integer;
  succ: stream;
begin
  poll
    inp?int(x) −> +succ;
      sort(succ, out); more := true;|
    inp?eos −> out!eos; more := false
  end;
  while more do
    poll
      inp?int(y) −>
        if x > y then
          begin succ!int(x); x := y end
        else succ!int(y)|
      inp?eos −> out!int(x);
        succ!eos; more := false
    end
end;
```

The sorting agents share a common output channel. Initially, an agent is the last one in the chain and is empty. After receiving the first value from its predecessor, the agent creates a successor and becomes non-empty. The agent now inputs the rest of the stream from its predecessor and keeps the smallest value x received so far. The rest it sends to its successor. When the agent inputs an *eos* signal it terminates as follows: if it is empty, the agent sends *eos* through the common channel; otherwise it outputs x on the common channel and sends *eos* to its successor.

As an example, while sorting the sequence

$$3,\ 1,\ 2,\ eos$$

the accumulator s starts as a single empty agent denoted by $< \phi >$ and is extended by a new agent for every value input:

Initially:	$s =< \phi >$
After inputting 3:	$s =< 3 >, < \phi >$
After inputting 1:	$s =< 1 >, < 3 >, < \phi >$
After inputting 2:	$s =< 1 >, < 2 >, < 3 >, < \phi >$

The sorting accumulator may be tested by means of a pipeline with three agents:

```
agent pipeline1;
use generate, sort, print;
var a, b: stream;
begin
  +a; +b; generate(a, 10, −1, 10);
  sort(a, b); print(b)
end;
```

The print agent accepts a stream and prints it.

The next pipeline merges two unordered streams, sorts the results, suppresses duplicates and prints the rest:

```
agent pipeline2;
use generate, merge, sort, suppress, print;
var a, b, c, d, e: stream;
begin
  +a; +b; +c; +d; +e;
  generate(a, 1, 1, 10);
  generate(b, 10, −1, 10);
  merge(a, b, c); sort(c, d);
  suppress(d, e); print(e)
end;
```

Example: prime sieve

A dynamic accumulator that inputs a finite sequence of natural numbers 1, 2, 3, ..., n and outputs those that are primes:

```
agent sieve(inp, out: stream);
var more: boolean; x, y: integer;
  succ: stream;
begin
  poll
    inp?int(x) −> +succ;
      sieve(succ, out); more := true|
    inp?eos −> out!eos; more := false
  end;
  while more do
    poll
      inp?int(y) −>
        if y mod x <> 0 then succ!int(y)|
      inp?eos −> out!int(x);
        succ!eos; more := false
    end;
end;
```

Initially, a sieve agent inputs a prime x from its predecessor and activates a successor. The agent then skips all further input which is divisible by x and sends the rest to its successor. At the end, the agent sends x through the common channel and sends *eos* either to its successor (if any) or through the output channel.

The sieve can be optimized somewhat by letting every agent output its prime as soon as it has been input. The present form of the algorithm was chosen to show that the sort and sieve agents are almost identical variants of the same programming paradigm. (They differ in one statement only!)

Since 2 is the only even prime, we may as well feed the sieve with odd numbers 3, 5, 7, ... only. The following pipeline prints all primes between 3 and 9999:

```
agent primes;
use generate, sieve, print;
var a, b: stream;
begin
  +a; +b; generate(a, 3, 2, 4999);
  sieve(a, b); print(b)
end;
```

Function evaluation

A function $f(x)$ can be evaluated by activating an agent with two parameters denoting the argument x and a channel. The agent evaluates $f(x)$, outputs the result on the channel and terminates.

A procedure can be implemented similarly.

Example: recursive Fibonacci

An agent that computes a Fibonacci number recursively by means of a tree of subagents:

```
type func = [val(integer)];

agent fibonacci(f: func; x: integer);
var g, h: func; y, z: integer;
begin
  if x <= 1 then f!val(x)
  else
    begin
      +g; fibonacci(g, x - 1);
      +h; fibonacci(h, x - 2);
      g?val(y); h?val(z); f!val(y + z)
    end
end;
```

Data representation

An agent can also implement a set of operations on a data representation.

Example: recursive set

Problem. Represent a set of integers as an agent with an input and an output channel. Initially, the set is empty. The set agent accepts three kinds of commands from a single user agent only:

1. Insert an integer n in the set:

   inp!insert(n)

2. Return a boolean b indicating if n is in the set:

inp!has(n); out?return(b)

3. Delete the set:

inp!delete

Solution.

```
type
  setinp = [insert(integer), has(integer), delete];
  setout = [return(boolean)];

agent intset(inp: setinp; out: setout);
type state = (empty, nonempty, deleted);
var s: state; x, y: integer; succ: setinp;
begin
  s := empty;
  while s = empty do
    poll
      inp?insert(x) -> +succ;
        intset(succ, out); s := nonempty|
      inp?has(x) -> out!return(false)|
      inp?delete -> s := deleted
    end;
  while s = nonempty do
    poll
      inp?insert(y) ->
        if x > y then
          begin succ!insert(x); x := y end
        else if x < y then succ!insert(y)|
      inp?has(y) ->
        if x >= y then out!return(x = y)
          else succ!has(y)|
      inp?delete -> succ!delete; s := deleted
    end
end;
```

The set agent is very similar to the sort and sieve agents. It contains either one member of the set or none. Initially, the agent is empty and answers false

to all membership queries. After the first insertion, it activates an empty successor to which it passes any command it cannot handle. To speed up processing, the set is ordered. Many insertions can proceed simultaneously in the pipeline. Insertion of an already existing member has no effect. A delete signal propagates through all the set agents and makes them terminate.

Monitors

A monitor is a scheduling agent that enables two or more user agents to share a resource. The user agents can invoke operations on the resource one at a time only. A monitor may use boolean expressions to delay operations until they are feasible.

Example: ring buffer

A monitor that implements a non-terminating ring buffer which can hold up to ten messages:

```
agent buffer(inp, out: stream);
const n = 10;
type contents = array [1..n] of integer;
var head, tail, length: integer;
  ring: contents;
begin
  head := 1; tail := 1; length := 0;
  while true do
    poll
      inp?int(ring[tail]) & length < n ->
        tail := tail mod n + 1;
        length := length + 1|
      out!int(ring[head]) & length > 0 ->
        head := head mod n + 1;
        length := length - 1
    end
end;
```

An empty buffer may input a message only. A full buffer may output only. When the buffer contains at least one and at most nine values, it is ready either to input or to output a message.

Example: scheduled printer

A monitor that gives one user agent at a time exclusive access to a printer during a sequence of write operations. The user agent must open the printer before writing and close it afterwards:

```
type printsym = [open, write(char), close];

agent printer(user: printsym);
var more: boolean; x: char;
begin
  while true do
    begin
      user?open; more := true;
      while more do
        poll
          user?write(x) -> print(x)|
          user?close -> more := false
        end
    end
end;
```

When the printer has received an open symbol from a user agent, it accepts only a (possibly empty) sequence of write symbols followed by a close symbol. This protocol prevents other agents from opening the printer and using it simultaneously. (The details of printing are ignored.)

Ring nets

So far, we have only considered agents connected by acyclic nets of channels. In the final example, the agents are connected by a cyclic net of channels.

Example: nim players

From a pile of 20 coins, three players take turns picking one, two or three coins from the pile. The player forced to pick the last coin loses the game.

The game is simulated by three agents connected by a ring of three channels. When the game begins, one of the agents receives all the coins:

```
agent nim;
use player;
var a, b, c: stream;
```

```
begin
  +a; +b; +c; player(20, a, b);
  player(0, b, c); player(0, c, a);
end;
```

The players behave as follows:

```
agent player(pile: integer;
  pred, succ: stream);
var more: boolean;
begin
  if pile > 0 then succ!int(pile − 1);
  more := true;
  while more do
    poll
      pred?int(pile) −>
        if pile > 1 then succ!int(pile − 1)
        else { loser }
          begin
            succ!eos; pred?eos; more := false
          end|
      pred!eos −> succ!eos; more := false
    end
end;
```

When an agent receives the pile from its predecessor, it reduces it and sends the rest (if any) to its successor. (To simplify the algorithm slightly, an agent always removes a single coin). The agent that picks the last coin sends *eos* to its successor and waits until the signal has passed through the other two agents and comes back from its predecessor. At that point, the loser terminates. When a non-losing agent receives *eos* instead of a pile, it passes the signal to its successor and terminates.

The dining philosophers problem (Hoare 1978) is another example of a ring net. It is left as an exercise to the reader.

4 DESIGN ISSUES

The following motivates some of the design decisions of Joyce.

Terminology and notation

In the literature, the word "process" often denotes a sequential process. Since a composite agent is not sequential, I prefer to use another word for communicating machines (namely, "agents").

It was tempting to use the notation of CSP (Hoare 1978) or one of the successors of Pascal, for example Modula-2 (Wirth 1982). However, in spite of its limitations, Pascal has a readable notation which is familiar to everyone. Chosing a pure Pascal subset has enabled me to concentrate on the concurrent aspects of Joyce.

Indirect naming

One of the major advantages of monitors is their ability to communicate with processes and schedule them without being aware of process names. Joyce agents also refer indirectly to one another by means of port variables.

In CSP, an input/output command must name the source or destination process directly. The text of a process must therefore be modified when it is used in different contexts. This complicates the examples in (Hoare 1978): the user of a process array $S(1..n)$ is itself named $S(0)$! And the prime sieve is composed of three different kinds of processes to satisfy the naming rules.

Direct process naming also makes it awkward to write a server with multiple clients of different kinds (such as the scheduled printer). If the clients are not known *a priori*, it is in fact impossible.

ECSP and RBCSP use process variables for indirect naming. CSP80, occam, Planet and a theoretical variant of CSP, which I shall call TCSP (Hoare 1985), use ports or channels.

Message declarations

So far, the most common errors in Joyce programs have been type errors in input/output commands. I am therefore convinced that any CSP language must include message declarations which permit complete type checking during compilation. In this respect, CSP and occam are insecure languages. Although ECSP does not include message declarations, the compiler performs type checking of messages after recognizing (undeclared) channels by statement analysis.

The simplest idea is to declare channels which can transfer messages of a single type only (as in CSP80 or Planet). But this does not even work well for a simple agent that copies a bounded stream. Such an agent needs

two channels, both capable of transferring two different kinds of symbols. Otherwise, four channels are required: two for stream values and two for *eos* signals.

As a modest increase in complexity, I considered a channel which can transfer messages of a finite number of distinct types $T_1, T_2, \ldots, T_n$. But this proposal is also problematic since (1) it is necessary to treat signals as distinct data types, and (2) an agent still needs multiple channels to distinguish between different kinds of messages of the same type (such as the *has* and *insert* symbols in the *intset* example).

To avoid a confusing proliferation of channels, the ability to define channel alphabets with named symbols seems essential. The symbol names play the same role as the (undeclared) "constructors" of CSP or the procedure names of monitors: they describe the nature of an event in which a process participates.

Channel sharing

The *intset* pipeline is made simpler and more efficient by the use of a single output channel shared by all the agents. A set agent which receives a query about the member it holds can immediately output the answer through the common channel instead of sending it through all its successors. This improvement was suggested in (Dijkstra 1982).

Channel sharing also simplifies the scheduled printer. If every channel can be used by two processes only, it is necessary to connect a resource process to multiple users by means of a quantifier called a "replicator."

I expect channel sharing to work well for lightly used resources. But, if a shared resource is used heavily, some user agents may be bypassed by others and thus prevented from using the resource. In such cases, it may be necessary to introduce separate user channels to achieve fairness.

Output polling

In CSP, ECSP, RBCSP and occam, polling is done by input commands only. This restriction prevents a sender and receiver from polling the same channel simultaneously. Unfortunately, it also makes the input and output of a ring buffer asymmetric (Hoare 1978).

Like CSP80 and TCSP, Joyce permits both input and output polling. It is the programmer's responsibility to ensure that a polling agent is always matched by an agent that executes an input/output statement. This prop-

erty is automatically satisfied in a hierarchical system in which every agent polls its masters only (Silberschatz 1979).

Polling loops

CSP includes a polling loop that terminates when all the processes polled have terminated. Hoare (1985) remarks: "The trouble with this convention is that it is complicated to define and implement."

In RBCSP, a process waiting for input from a terminated process is terminated only when all processes are waiting or terminated.

A Joyce agent terminates when it reaches the end of its procedure. This is a much more flexible mechanism which enables an agent to send a termination signal to another agent without terminating itself.

I resisted the temptation to include polling loops, such as

do inp?int(x) −> out!int(x)
until inp?eos −> out!eos **end**

Although this simplifies the copy and printer agents, it cannot be used directly in the other examples. It may even complicate programs, if it is used where it is inappropriate.

Unbounded activation

In CSP one can activate only a fixed number of processes simultaneously. If these processes terminate, they do it simultaneously. A process cannot activate itself recursively. It is, however, possible to activate a fixed-length array of indexed processes which can imitate the behavior (but not quite the elegance) of a recursive process.

Joyce supports unbounded (recursive) agent activation. The beauty of the recursive algorithms is sufficient justification for this feature. The ability to activate identical agents by iteration and recursion removes the need for indexed agents (as in CSP, RBCSP, Planet and occam). The rule that an agent terminates only when all its subagents have terminated was inspired by the task concept of Ada (Roubine 1980).

Procedures and functions

To force myself to make agents as general as possible, I excluded ordinary procedures and functions from Joyce. As a result, I felt obliged to design an agent concept which includes the best features of Pascal procedures: value

parameters, recursion and efficient implementation. Although agent procedures may be recursive, every agent has one instance only of its own variables. Consequently, a compiler can determine the lengths of agent activation records. This simplifies storage allocation considerably.

Security

A programming language is secure if its compiler and run-time support can detect all violations of the language rules (Hoare 1973). Programs written in an insecure language may cause obscure system-dependent errors which are inexplicable in terms of the language report. Such errors can be extremely difficult to locate and correct.

Joyce is a far more secure language than Pascal (Welsh 1977). A compiler can check message types and ensure that agents use disjoint sets of variables only. (The disjointness is automatically guaranteed by the syntax and scope rules.)

When an agent is activated, every word of its activation record may be set to nil. Afterwards a simple run-time check can detect unitialized port variables.

There are no dangling references, either, to channels that have ceased to exist. Every port variable of an agent is either nil or points to an internal or external channel of the agent. Now, an internal channel exists as long as the agent and its port variables exist. And an external channel exists as long as the ancestor that created it. This ancestor, in turn, exists at least as long as the given agent. So, a port variable is either nil or points to an existing channel.

Implementability

The first Joyce compiler is a Pascal program of 3300 lines which generates P-code. The code is currently interpreted by a Pascal program of 1000 lines. (Reals are not implemented yet.) The surprisingly simple implementation of agents and channels will be described in a future paper.

Proof rules

The problems of finding proof rules for Joyce are currently being studied and are not discussed here. However, the algorithms shown have a convincing simplicity that makes me optimistic in this respect.

Language comparison

Table 1 summarizes the key features of the CSP languages (except TCSP).

Table 1

	CSP	occam	ECSP	Planet	RBCSP	CSP80	Joyce
Indirect naming	−	+	+	+	+	+	+
Message declaration	−	−	−	+	+	+	+
Input polling	+	+	+	−	+	+	+
Output polling	−	−	−	−	−	+	+
Recursion	−	−	−	−	−	−	+

Hoare (1978) emphasized that CSP should not be regarded as suitable for use as a programming language but only as a partial solution to the problems tackled. However, all that remained to be done was to modify these concepts. CSP is still the foundation for the new generation of concurrent programming languages discussed here.

5 FINAL REMARKS

This paper has presented a secure programming language which removes several restrictions of the original CSP proposal by introducing:

1. port variables
2. channel alphabets
3. output polling
4. channel sharing
5. recursive agents

The language has been implemented on a personal computer.

More work remains to be done on verification rules and implementation of the language on a parallel computer. The language needs to be used extensively for the design of parallel algorithms before a final evaluation can be made.

Acknowledgements

It is a pleasure to acknowledge the helpful comments of Birger Andersen, Peter T. Andersen, Lotte Bangsborg, Peter Brinch, Niels Christian Juul and Bo Salomon.

References

Baiardi, F., Ricci, L., and Vanneschi, M. 1984. Static checking of interprocess communication in ECSP. *ACM SIGPLAN Notices 19*, 6 (June), 290–299.

Brinch Hansen, P. 1973. *Operating System Principles.* Prentice-Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. 1975. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering 1*, 2 (June), 199–205.

Brinch Hansen, P. 1976. The Solo operating system. *Software—Practice and Experience 6*, 2 (April–June), 141–205.

Crookes, D., and Elder, J.W.G. 1984. An experiment in language design for distributed systems. *Software–Practice and Experience 14*, 10 (October), 957–971.

Dijkstra, E.W. 1982. *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, New York, 147–160.

Hoare, C.A.R. 1973. Hints on programming language design. Computer Science Department, Stanford University, Stanford, CA, (December).

Hoare, C.A.R. 1974. Monitors: An operating system structuring concept. *Communications of the ACM 17*, 10 (October), 549–557.

Hoare, C.A.R. 1978. Communicating sequential processes. *Communications of the ACM 21*, 8 (August), 666–677.

Hoare, C.A.R. 1985. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ.

Holt, R.C. 1982. A short introduction to Concurrent Euclid. *ACM SIGPLAN Notices 17*, (May), 60–79.

Inmos, Ltd. 1984. *occam Programming Manual.* Prentice-Hall, Englewood Cliffs, NJ.

Jazayeri, M., Ghezzi, C., Hoffman, D., Middleton, D., and Smotherman, M. 1980. CSP/80: A language for communicating sequential processes. *IEEE Compcon Fall*, (September), 736–740.

Lampson, B.W., and Redell, D.D. 1980. Experience with processes and monitors in Mesa. *Communications of the ACM 23*, 2 (February), 105–117.

Roper, T.J., and Barter, C.J. 1981. A communicating sequential process language and implementation. *Software—Practice and Experience 11*, 11 (November), 1215–1234.

Roubine, O., and Heliar, J.–C. 1980. Parallel processing in Ada. In *On the Construction of Programs*, R.M. McKeag, and A.M. Macnaghten, Eds. Cambridge University Press, Cambridge, 193–212.

Silberschatz, A. 1979. Communication and synchronization in distributed systems. *IEEE Transactions on Software Engineering 5*, 6 (November), 542–546.

Welsh, J., Sneeringer, W.J., and Hoare, C.A.R. 1977. Ambiguities and insecurities in Pascal. *Software—Practice and Experience 7*, 6 (November–December), 685–696.

Welsh, J., and Bustard, D.W. 1979. Pascal-Plus—Another language for modular multiprogramming. *Software—Practice and Experience 9*, 11 (November), 947–957.

Wirth, N. 1971. The programming language Pascal. *Acta Informatica 1*, 35–63.

Wirth, N. 1977. Modula—A language for modular multiprogramming. *Software—Practice and Experience 7*, 1 (January–February), 3–35.

Wirth, N. 1982. *Programming in Modula-2.* Springer-Verlag, New York.